Nix with R and devtools

Rohit Goswami · 2020-06-06 · tools · nix · workflow · R

> *This post discusses briefly, the `nix-shell` environment for reproducible programming. In particular, there is an emphasis on extensions for installing and working with packages not in CRAN, i.e. packages off Github which are normally installed with `devtools`.*

Background

The entire nix ecosystem is fantastic, and is the main packaging system used by d-SEAMS as well. Recently I began working through the excellent second edition of "Statistical Rethinking" by Richard McElreath[^fn:1].

Unfortunately, the `rethinking` package which is a major component of the book itself depends on the V8 engine for some reason. The reigning AUR[^fn:2] package (V8-r) broke with a fun error message I couldn't be bothered to deal with. Ominously, the rest of the logs prominently featured `Warning: Running gclient on Python 3.`. Given that older `python` versions have been permanently retired, this seemed like a bad thing to deal with[^fn:3]. In any case, having weaned off non-nix dependency tools for `python` and friends, it seemed strange to not do the same for R.

The standard installation for the package entails obtaining `rstan` (which is trivial with `nixpkgs`) and then using:

```
install.packages(c("coda","mvtnorm","devtools","loo","dagitty"))
library(devtools)
devtools::install_github("rmcelreath/rethinking")
```

We will break this down and work through this installation in Nix space.

Nix and R

The standard approach to setting up a project `shell.nix` is simply by using the `mkshell` function. There are some common aspects to this workflow, with more language specific details documented here. A simple first version might be:

```
let
    pkgs = import <nixpkgs> { };
in pkgs.mkShell {
    buildInputs = with pkgs; [
        zsh
        R
        rPackages.ggplot
        rPackages.data_table
    ];
    shellHook = ''
    echo "hello"
    '';
  LOCALE_ARCHIVE = stdenv.lib.optionalString stdenv.isLinux
    "${glibcLocales}/lib/locale/locale-archive";
}
```

Here we can install CRAN packages as easily as regular packages (like R), except for the fact that they are kept in a `pkgs.rPackages` environment, as opposed to `pkgs`. This is actually a common convention most languages with central repos. The most interesting thing to note is that, similar to the convention for `nix-python` setups, packages with a dot in the name will be converted to an underscore, i.e. `data.table` -> `data_table`.

However, for the rethinking package, and many others, there is no current CRAN package, and so the `rPackages` approach fails.

The `LOCALE_ARCHIVE` needs to be set for Linux machines, and is required for working with other packages.

Nix-R and Devtools

To work with non-CRAN packages, we need to modify our package setup a little. We will also simplify our file to split the `pkgs` and the `r-pkgs`.

Naive Approach

The naive approach works by using the `shellHook` to set `R_LIBS_USER` to save user packages per-project.

```
{ pkgs ? import <nixpkgs> { } }:
with pkgs;
let
  my-r-pkgs = rWrapper.override {
    packages = with rPackages; [
      ggplot2
      knitr
      rstan
      tidyverse
      V8
      dagitty
      coda
      mvtnorm
      shape
      Rcpp
      tidybayes
    ];
  };
in mkShell {
  buildInputs = = with pkgs;[ git glibcLocales openssl openssh curl wget ];
  inputsFrom = [ my-r-pkgs ];
  shellHook = ''
    mkdir -p "$(pwd)/_libs"
    export R_LIBS_USER="$(pwd)/_libs"
  '';
  GIT_SSL_CAINFO = "${cacert}/etc/ssl/certs/ca-bundle.crt";
  LOCALE_ARCHIVE = stdenv.lib.optionalString stdenv.isLinux
    "${glibcLocales}/lib/locale/locale-archive";
}
```

Note that here we will also need to set the `GIT_SSL_CAINFO` to prevent some errors during the build process[^fn:4].

Native Approach

The native approach essentially leverages the `nix` method for building `R` packages. This is the most reproducible of the lot, and also has the useful property of storing the files in the `nix-store` so re-using packages across different projects will not store, build or download the package again. The values required can be calculated from `nix-prefetch-git` as follows:

```
nix-env -i nix-prefetch-git
nix-prefetch-git https://github.com/rmcelreath/rethinking.git
```

The crux of this approach is the following snippet[^fn:5]:

```
(buildRPackage {
  name = "rethinking";
  src = fetchFromGitHub {
    owner = "rmcelreath";
    repo = "rethinking";
    rev = "d0978c7f8b6329b94efa2014658d750ae12b1fa2";
    sha256 = "1qip6x3f6j9lmcmck6sjrj50a5azqfl6rfhp4fdj7ddabpb8n0z0";
  };
  propagatedBuildInputs = [ coda MASS mvtnorm loo shape rstan dagitty ];
 })
```

Project Shell

This formulation for some strange reason does not work from the shell or environment by default, but does work with `nix-shell --run bash --pure`.

```
{ pkgs ? import <nixpkgs> { } }:
with pkgs;
let
  my-r-pkgs = rWrapper.override {
    packages = with rPackages; [
      ggplot2
      knitr
      rstan
      tidyverse
      V8
      dagitty
      coda
      mvtnorm
      shape
      Rcpp
      tidybayes
      (buildRPackage {
        name = "rethinking";
        src = fetchFromGitHub {
          owner = "rmcelreath";
          repo = "rethinking";
          rev = "d0978c7f8b6329b94efa2014658d750ae12b1fa2";
          sha256 = "1qip6x3f6j9lmcmck6sjrj50a5azqfl6rfhp4fdj7ddabpb8n0z0";
        };
        propagatedBuildInputs = [ coda MASS mvtnorm loo shape rstan dagitty ];
      })
    ];
  };
in mkShell {
  buildInputs = with pkgs; [ git glibcLocales openssl which openssh curl wget my-r-pkgs ];
  shellHook = ''
    mkdir -p "$(pwd)/_libs"
    export R_LIBS_USER="$(pwd)/_libs"
    echo ${my-r-pkgs}/bin/R
  '';
  GIT_SSL_CAINFO = "${cacert}/etc/ssl/certs/ca-bundle.crt";
  LOCALE_ARCHIVE = stdenv.lib.optionalString stdenv.isLinux
    "${glibcLocales}/lib/locale/locale-archive";
}
```

The reason behind this is simply that `rWrapper` forms an extra package which has lower precedence than the user profile `R`, which is documented in more detail here on the NixOS wiki.

User Profile

This is a more general approach which defines the environment for R with all the relevant libraries and is described in the nixpkgs manual. The following code should be placed in `$HOME/.config/nixpkgs/config.nix`:

```
{
  packageOverrides = super:
    let self = super.pkgs;
    in {
      rEnv = super.rWrapper.override {
        packages = with self.rPackages; [
          ggplot2
          knitr
          tidyverse
          tidybayes
          (buildRPackage {
            name = "rethinking";
            src = self.fetchFromGitHub {
              owner = "rmcelreath";
              repo = "rethinking";
              rev = "d0978c7f8b6329b94efa2014658d750ae12b1fa2";
              sha256 = "1qip6x3f6j9lmcmck6sjrj50a5azqfl6rfhp4fdj7ddabpb8n0z0";
            };
            propagatedBuildInputs =
              [ coda MASS mvtnorm loo shape rstan dagitty ];
          })
        ];
      };
    };
}
```

This snippet allows us to use our `R` as follows:

```
# Install things
nix-env -f "<nixpkgs>" -iA rEnv
# Fix locale
export LOCALE_ARCHIVE="$(nix-build --no-out-link "<nixpkgs>" -A glibcLocales)/lib/locale/locale-archive"
# Profit
R
```

Note that in this method, on Linux systems, the locale problem has to be fixed with the explicit export. This means that this should be used mostly with project level environments, instead of populating the global shell RC files.

Update: There is another post with methods to reload this configuration automatically

Conclusions

Of the methods described, the most useful method for working with packages not hosted on CRAN is through the user-profile, while the `shell.nix` method is useful in conjunction, for managing various projects. So the ideal approach is then to use the user profile for installing anything which normally uses `devtools` and then use `shell.nix` for the rest.

Note that if the Project Shell is used with a User Profile as described in the next section, all packages defined there can be dropped and then the project shell does not need to execute `R` by default. The simplified `shell.nix` is then simply:

```
{ pkgs ? import <nixpkgs> { } }:
with pkgs;
let
  my-r-pkgs = rWrapper.override {
    packages = with rPackages; [
      ggplot2
    ];
  };
in mkShell {
  buildInputs = with pkgs;[ git glibcLocales openssl which openssh curl wget my-r-pkgs ];
  inputsFrom = [ my-r-pkgs ];
  shellHook = ''
    mkdir -p "$(pwd)/_libs"
    export R_LIBS_USER="$(pwd)/_libs"
  '';
  GIT_SSL_CAINFO = "${cacert}/etc/ssl/certs/ca-bundle.crt";
  LOCALE_ARCHIVE = stdenv.lib.optionalString stdenv.isLinux
    "${glibcLocales}/lib/locale/locale-archive";
}
```

The entire workflow for `rethinking` is continued here.